Exploring Python Functions

Estimated time needed: 15 minutes

Objectives:

By the end of this reading, you should be able to:

1. Describe the function concept and the importance of functions in programming
2. Write a function that takes inputs and performs tasks
3. Use built-in functions like len(), sum(), and others effectively
4. Define and use your functions in Python
5. Differentiate between global and local variable scopes
6. Use loops within the function
7. Modify data structures using functions

Introduction to functions

A function is a fundamental building block that encapsulates specific actions or computations. As in mathematics, where functions take inputs and produce outputs, programming functions perform similarly. They take inputs, execute predefined actions or calculations, and then return an output.

Purpose of functions

Functions promote code modularity and reusability. Imagine you have a task that needs to be performed multiple times within a program. Instead of duplicating the same code at various places, you can define a function once and call it whenever you need that task. This reduces redundancy and makes the code easier to manage and maintain.

Benefits of using functions

Modularity: Functions break down complex tasks into manageable components
Reusability: Functions can be used multiple times without rewriting code
Readability: Functions with meaningful names enhance code understanding
Debugging: Isolating functions eases troubleshooting and issue fixing
Abstraction: Functions simplify complex processes behind a user-friendly interface
Collaboration: Team members can work on different functions concurrently
Maintenance: Changes made in a function automatically apply wherever it's used

How functions take inputs, perform tasks, and produce outputs

Inputs (Parameters)

Functions operate on data, and they can receive data as input. These inputs are known as *parameters* or *arguments*. Parameters provide functions with the necessary information they need to perform their tasks. Consider parameters as values you pass to a function, allowing it to work with specific data.

Performing tasks

Once a function receives its input (parameters), it executes predefined actions or computations. These actions can include calculations, operations on data, or even more complex tasks. The purpose of a function determines the tasks it performs. For instance, a function could calculate the sum of numbers, sort a list, format text, or fetch data from a database.

Producing outputs

After performing its tasks, a function can produce an output. This output is the result of the operations carried out within the function. It's the value that the function "returns" to the code that called it. Think of the output as the end product of the function's work. You can use this output in your code, assign it to variables, pass it to other functions, or even print it out for display.

Example:

Consider a function named `calculate_total` that takes two numbers as input (parameters), adds them together, and then produces the sum as the output. Here's how it works:

```
    def calculate_total(a, b):  # Parameters: a and b
        total = a + b           # Task: Addition
        return total            # Output: Sum of a and b
    result = calculate_total(5, 7)  # Calling the function with inputs 5 and 7
    print(result)  # Output: 12
```

Python's built-in functions

Python has a rich set of built-in functions that provide a wide range of functionalities. These functions are readily available for you to use, and you don't need to be concerned about how they are implemented internally. Instead, you can focus on understanding what each function does and how to use it effectively.

Using built-in functions or Pre-defined functions

To use a built-in function, you simply call the function's name followed by parentheses. Any required arguments or parameters are passed into the function within these parentheses. The function then performs its predefined task and may return an output you can use in your code.

Here are a few examples of commonly used built-in functions:

len(): Calculates the length of a sequence or collection

```
    string_length = len("Hello, World!")  # Output: 13
    list_length = len([1, 2, 3, 4, 5])  # Output: 5
```

sum(): Adds up the elements in an iterable (list, tuple, and so on)

```
    total = sum([10, 20, 30, 40, 50])  # Output: 150
```

max(): Returns the maximum value in an iterable

```
    highest = max([5, 12, 8, 23, 16])  # Output: 23
```

min(): Returns the minimum value in an iterable

```
    lowest = min([5, 12, 8, 23, 16])  # Output: 5
```

Python's built-in functions offer a wide array of functionalities, from basic operations like len() and sum() to more specialized tasks.

Defining your functions

Defining a function is like creating your mini-program:

1. Use `def` followed by the function name and parentheses

Here is the syntax to define a function:

```
    def function_name():
        pass
```

A "`pass`" statement in a programming function is a placeholder or a no-op (no operation) statement. Use it when you want to define a function or a code block syntactically but do not want to specify any functionality or implementation at that moment.

- **Placeholder:** "`pass`" acts as a temporary placeholder for future code that you intend to write within a function or a code block.
- **Syntax Requirement:** In many programming languages like Python, using "`pass`" is necessary when you define a function or a conditional block. It ensures that the code remains syntactically correct, even if it doesn't do anything yet.
- **No Operation:** "`pass`" itself doesn't perform any meaningful action. When the interpreter encounters "pass", it simply moves on to the next statement without executing any code.

Function Parameters:

- Parameters are like inputs for functions
- They go inside parentheses when defining the function
- Functions can have multiple parameters

Example:

```
    def greet(name):
        print("Hello, " + name)
    result = greet("Alice")
    print(result)  # Output: Hello, Alice
```

Docstrings (Documentation Strings)

- Docstrings explain what a function does
- Placed inside triple quotes under the function definition
- Helps other developers understand your function

Example:

```
    def multiply(a, b):
        """
        This function multiplies two numbers.
        Input: a (number), b (number)
        Output: Product of a and b
        """
        print(a * b)
    multiply(2,6)
```

Return statement

- Return gives back a value from a function
- Ends the function's execution and sends the result
- A function can return various types of data

Example:

```
    def add(a, b):
        return a + b
    sum_result = add(3, 5)  # sum_result gets the value 8
```

Understanding scopes and variables

Scope is where a variable can be seen and used:

- **Global Scope:** Variables defined outside functions; accessible everywhere
- **Local Scope:** Variables inside functions; only usable within that function

Example:

Part 1: Global variable declaration

```
global_variable = "I'm global"
```

This line initializes a global variable called `global_variable` and assigns it the value "I'm global".

Global variables are accessible throughout the entire program, both inside and outside functions.

Part 2: Function definition

```
def example_function():
    local_variable = "I'm local"
    print(global_variable)  # Accessing global variable
    print(local_variable)   # Accessing local variable
```

Here, you define a function called `example_function()`.

Within this function:

- A local variable named local_variable is declared and initialized with the string value "I'm local." This variable is local to the function and can only be accessed within the function's scope.
- The function then prints the values of both the **global variable (global_variable) and the local variable (local_variable)**. It demonstrates that you can access global and local variables within a function.

Part 3: Function call

```
example_function()
```

In this part, you call the `example_function()` by invoking it. This results in the function's code being executed.
As a result of this function call, it will print the values of the global and local variables within the function.

Part 4: Accessing global variable outside the function

```
print(global_variable)  # Accessible outside the function
```

After calling the function, you print the value of the global variable `global_variable` outside the function. **This demonstrates that global variables are accessible inside and outside of functions.**

Part 5: Attempting to access local variable outside the function

```
# print(local_variable)  # Error, local variable not visible here
```

In this part, you are attempting to print the value of the local variable `local_variable` outside of the function. However, this line would result in an error.

Local variables are only visible and accessible within the scope of the function where they are defined.

Attempting to access them outside of that scope would raise a "`NameError`".

Using functions with loops

Functions and loops together

1. Functions can contain code with loops
2. This makes complex tasks more organized
3. The loop code becomes a repeatable function

Example:

```
def print_numbers(limit):
    for i in range(1, limit+1):
        print(i)
print_numbers(5)  # Output: 1 2 3 4 5
```

Enhancing code organization and reusability

1. Functions group similar actions for easy understanding
2. Looping within functions keeps code clean
3. You can reuse a function to repeat actions

Example

```
def greet(name):
    return "Hello, " + name
for _ in range(3):
    print(greet("Alice"))
```

Modifying data structure using functions

You'll use Python and a list as the data structure for this illustration. In this example, you will create functions to add and remove elements from a list.

Part 1: Initialize an empty list

```
# Define an empty list as the initial data structure
my_list = []
```

In this part, you start by creating an empty list named `my_list`. This empty list serves as the data structure that you will modify throughout the code.

Part 2: Define a function to add elements

```
# Function to add an element to the list
def add_element(data_structure, element):
    data_structure.append(element)
```

Here, you define a function called `add_element`. This function takes two parameters:

- `data_structure`: This parameter represents the list to which you want to add an element
- `element`: This parameter represents the element you want to add to the list

Inside the function, you use the `append` method to add the provided element to the data_structure, which is assumed to be a list.

Part 3: Define a function to remove elements

```
# Function to remove an element from the list
def remove_element(data_structure, element):
    if element in data_structure:
        data_structure.remove(element)
    else:
        print(f"{element} not found in the list.")
```

In this part, you define another function called `remove_element`. It also takes two parameters:

- `data_structure`: The list from which we want to remove an element
- `element`: The element we want to remove from the list

Inside the function, you use conditional statements to check if the element is present in the data_structure. If it is, you use the `remove` method to remove the first occurrence of the element. If it's not found, you print a message indicating that the element was not found in the list.

Part 4: Add elements to the list

```
# Add elements to the list using the add_element function
add_element(my_list, 42)
add_element(my_list, 17)
add_element(my_list, 99)
```

Here, you use the `add_element` function to add three elements (42, 17, and 99) to the `my_list`. These are added one at a time using function calls.

Part 5: Print the current list

```
# Print the current list
print("Current list:", my_list)
```

This part simply prints the current state of the `my_list` to the console, allowing us to see the elements that have been added so far.

Part 6: Remove elements from the list

```
# Remove an element from the list using the remove_element function
remove_element(my_list, 17)
remove_element(my_list, 55)  # This will print a message since 55 is not in the list
```

In this part, you use the `remove_element` function to remove elements from the my_list. First, you attempt to remove 17 (which is in the list), and then you try to remove 55 (which is not in the list). **The second call to remove_element will print a message indicating that 55 was not found.**

Part 7: Print the updated list

```
# Print the updated list
print("Updated list:", my_list)
```

Finally, you print the updated `my_list` to the console. This allows us to observe the modifications made to the list by adding and removing elements using the defined functions.

Conclusion

Congratulations! You've completed the Reading Instruction Lab on Python functions. You've gained a solid understanding of functions, their significance, and how to create and use them effectively. These skills will empower you to write more organized, modular, and powerful code in your Python projects.